Designing A Low-Latency Cuckoo Hash Table for Write-Intensive Workloads Using RDMA

Tyler Szepesi Bernard Wong Ben Cassell Tim Brecht
School of Computer Science, University of Waterloo
{stszepesi, bernard, becassel, brecht}@cs.uwaterloo.ca

ABSTRACT

In this paper, we present Nessie, a low-latency cuckoo hash table design that uses only one-sided RDMA operations to perform read and write requests. Nessie makes use of a self-verifying data-structure to handle reads that occur in parallel to writes, and atomic RDMA compare-and-swap operations to apply multiple operations with at most one data modification as a single atomic unit without explicit locking. In order to perform key migration, a relatively common housekeeping operation in cuckoo hash tables, using only one-sided RDMA operations, a Nessie client inserts an incomplete data entry that can be overwritten but not read, allowing it to reveal the entry only after the key migration succeeds. We analytically show that, with a reasonable hash table load factor, Nessie has lower latency than a single TCP roundtrip time and can complete write requests in less than 25 microseconds. Finally, we propose extending RDMA to support Remote Hardware Transactional Memory and outline how it can be used to greatly simplify Nessie's design.

1. INTRODUCTION

Distributed in-memory datastores are increasingly being used to either augment or replace traditional databases. By avoiding slow disk and flash drive accesses, they can handle orders of magnitude higher request rates. As a result, the performance bottleneck of data-intensive and latency-sensitive datacenter applications (such as realtime analytics and high-performance scientific computing) shifts from the storage device to the network. Traditional TCP/IP-based network communication introduces a significant amount of latency [13] and requires processing by the storage host's CPU to handle every request. At high data-rates, a storage host has to dedicate a significant amount of its processing capacity to handling just basic read and write requests [11].

Past work [15, 11, 3] has looked at using RDMA to improve the performance of distributed in-memory datastores. RDMA is a set of operations available to many high-performance network adapters (NICs) that provides clients with direct memory access to a remote server. By using self-verifying data structures that can detect corrupted reads due to concurrent writes [11, 3], a client can use one-sided RDMA read operations to perform a remote read request without involving the storage host's CPU. However, in existing systems, write requests must still be handled by a process on the storage host, and RDMA is only used as a lower latency substitute for TCP/IP [15, 11, 3]. Therefore, these systems only provide partial solutions for write-intensive workloads such as a distributed caching layer serving many concurrently writing clients.

The key challenge to using efficient one-sided RDMA operations for writes is in providing mutual exclusion to shared memory with minimal coordination overhead between the NIC and main CPU. Unlike remote read requests, where corrupted reads can be detected efficiently by the client and reissued, a corrupted write due to concurrent remote writers can be expensive to detect, and failing to detect a corruption results in data loss. Furthermore, data insert and delete requests may, depending on the design of the in-memory datastore, require additional changes to the storage data-structure. For example, in a cuckoo hash table [12, 4], an insert request may require additional data movement of hash-colliding keys. An efficient design should allow a remote client to perform these relatively common housekeeping operations using one-sided RDMA operations.

In this paper, we present Nessie, an RDMA-optimized cuckoo hash table design that uses only one-sided RDMA operations for performing both read and write requests. Like previous work, Nessie stores data using self-verifying data-structures to avoid reading corrupted data. Nessie, however, uses atomic RDMA compare-and-swap (CAS) operations and key-specific version numbers to ensure that multiple one-sided RDMA operations in a request with at most one data modification apply atomically. If this is not possible, the modification is rolled back and the request is reissued. Although a lock-based approach would be easy to implement, it would suffer from a variety of undesirable properties. Clients that fail while holding a lock, for instance, would result in deadlocks. Nessie therefore does not use CAS operations to build locks.

Nessie also allows clients to perform key migration, a relatively common housekeeping operation in cuckoo hash tables, using only one-sided RDMA operations. The main challenge with key migration is the need to atomically perform both an insert and delete request, which involves two data modifications. Nessie simulates atomicity by inserting incomplete entries into the table that can be overwritten but not read until an additional flag is later toggled. This enables Nessie to delay revealing the results of data modifications until it can ensure that the key migration does not need to be rolled back.

Although Nessie is able to provide low-latency reads and writes using one-sided RDMA operations, its design is significantly more complex than other hash tables. Even minor changes to the design would require substantial effort to reason about the correctness of the changes. To reduce this complexity, we propose extending RDMA to support *Remote*

Hardware Transactional Memory (RHTM) in rack-scale systems, and outline a version of Nessie that uses RHTM.

Overall, this paper makes three contributions:

- We present Nessie, a novel cuckoo hash table design that uses one-sided RDMA operations for both read and write requests.
- We analytically show that, with a reasonable hash table load factor of 0.5, Nessie completes read requests in 2 roundtrips and the majority of write requests in less than 5.8 roundtrips.
- We describe a new RHTM protocol which greatly simplifies Nessie's implementation.

2. BACKGROUND AND RELATED WORK

There are two main attractions for using one-sided RDMA operations: zero-copy data transfer, and CPU bypassing. Zero-copy data transfer refers to the fact that RDMA operations do not require making a copy of the application's data. For example, sending data from a client to a server over TCP/IP first requires the data to be copied from the client application's memory into the client kernel's memory. The data is then transferred across the network to the server kernel's memory, from which it is copied into the server application's memory. Alternatively, sending data from a client to a server using RDMA directly moves the data from the client application's memory to the server application's memory, with no buffering in the kernel.

Although it is possible to reduce the amount of copying done by using operations such as sendfile and mmap, the kernel is nevertheless responsible for handling the data transfer. The problem with this approach is that it requires the CPU to be utilized while performing a data transfer. With one-sided RDMA operations, both the kernel and CPU are bypassed by having the NIC manage the transfer of data, thereby freeing the CPU to perform other operations.

The two simplest one-sided RDMA operations are read and write. A RDMA read fetches the data from a memory region on the server into a memory region on the client. Conversely, a RDMA write places data from a local memory region on the client into a remote memory region on the server. RDMA also provides an atomic compare-and-swap (CAS) operation. The CAS operation takes three 64 bit parameters and compares the value of first parameter with the value stored at the remote memory region specified by the second parameter. If the values are equal, the value of the third parameter is written into the remote address. Otherwise, the remote memory address is left unchanged. In either case, the value stored at the remote memory address is returned as the result of the operation.

Figure 1 demonstrates the latency of one-sided RDMA operations against that of traditional TCP/IP send and receive. The experiments are performed using 40 GbE with RoCE [2] to allow for RDMA over Ethernet. TCP/IP has an order of magnitude higher latency than any RDMA operation even when there is no contention for CPU resources. The results also show that, because of the high bandwidth network, there is negligible difference in latency between a 10B and a 1KB data transfer.

RDMA is not a new technology, but it has received renewed interest in recent literature due to its current trend towards affordability. One common use for RDMA is to improve the performance of file and storage systems [5, 6, 7]. RDMA has also seen use as a communication mechanism, such as implementations of MPI that leverage RDMA [9, 10, 14].

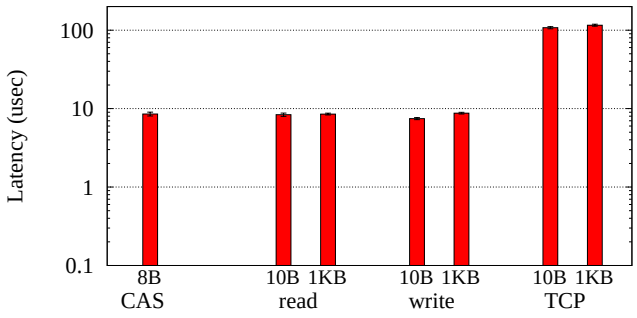

Figure 1: RDMA Latency

In this work, we are exploring the use of one-sided RDMA for key-value storage, which has been proposed by other systems such as Pilaf [11] and FaRM [3]. Pilaf is a key-value storage system that uses one-sided RDMA reads to access data that is *self-verifying* and leaves write operations to the server. If the server is writing data at the same time that a client is reading the data, the client recognizes this by comparing the data against the embedded hash to verify the correctness of the read. If a client want to perform a write in Pilaf, it communicates this via an RDMA send to the server. FaRM makes a similar design decision but relies on guarantees made by the hardware to allow the clients to verify the correctness of the read. FaRM provides an alternative communication primitive for writes that allows messages to be exchanged between client and server using only one-sided RDMA operations.

While it is clear from past work that RDMA is a powerful tool for improving the performance of key-value storage, previous systems still rely on the server to handle data modifications. In this work, we explore the natural, if complex, next step of leveraging one-sided RDMA write and atomic compare-and-swap operations to allow the clients to manage the data manipulation of the hash table without locks or interaction with the server.

3. DESIGN

There are two parts to a Nessie hash table: the data table (D) and the index table (I). The data table is made up of data items, and is where the actual key-value pairs are stored. The data items are stored as a contiguous block of memory which is accessed via RDMA read or write operations. Because the read operations are not atomic, and read/write conflicts are possible when a data item is recycled, we store a hash of the data alongside the data itself so that the client can verify its correctness. Additionally, the data item may be valid according to the hash, but have a flag marking the data as invalid (which is set during a write operation before it is complete). If, during a read (and only during a read), the data is deemed invalid, the read is repeated with exponential backoff to minimize further invalid reads.

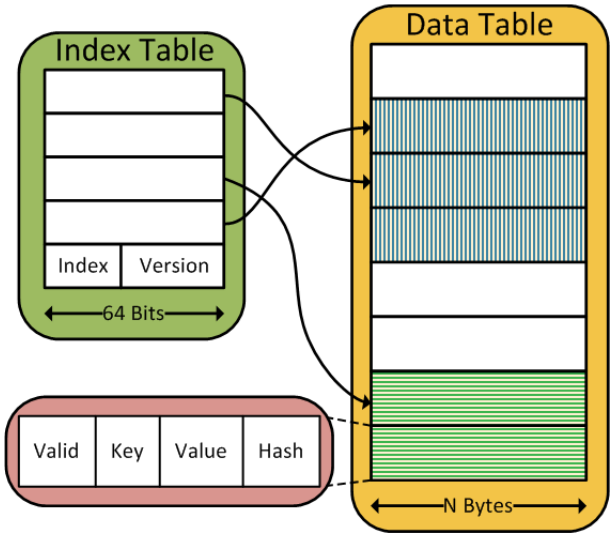

Figure 2: A Sample Nessie Hash Table

The index table consists of entries containing both an index and a version number. The index is the location in the data table at which the data item for the corresponding hash slot is stored. An initial index value of 0 is used to indicate that the slot is empty. The version number is used to prevent synchronization issues such as the ABA problem [1] which could occur if a client reuses a data entry in the data table. All access to the index table is done using the CAS operation to ensure that the index retrieved is done so atomically (and the retrieved value is therefore concurrency-safe). Figure 2 shows a simplified view of a Nessie hash table.

To avoid multiple writers concurrently writing to the same data item, each client is given ownership over a set of data items and clients may only update data items that they own. When a client wants to write a value to the hash table, they first write the data to a free data item which they own (a data item that is not referenced by the index table) with a flag marking the data as invalid. Once the write operation is complete the index table is updated, and the data is made valid.

After a data item has been deleted or replaced by a new write, the client that owns that data item may reuse it for future write operations. In the simplest case, the client that owns the data item is also the client that removes the reference to the data item from the index table. If the client that deletes the data item is not the owner, it must inform the owning client that the data item is no longer in use. At this point, the data item can be used for a future write operation.

The high-level structure of every operation on the hash table consists of: reading the primary and secondary indices for the specified key; performing the operation on the appropriate items at the index in the data table; checking that the hash locations have not been modified by another client using CAS on entries of the index table; and setting the valid bits on any new data items. If at any point during the operation it is detected that another client is operating on the same hash location, the operation is halted and retried with exponential backoff. Notice that halting during a write operation could leave the hash table with an active and invalid data item. This is acceptable as long as the client's operation eventually completes because any read operations for this key will notice either a hash mismatch or an unset valid bit. In either case, the operation retries with exponential backoffs until the write succeeds. If a client fails, it is the responsibility of the server to detect the failure, and clean up any invalid state left behind by that client using a periodic background process.

Because both the write and delete operations end the same way, we describe the process of completing either operation as a function, $finish$. The $finish$ function takes three arguments: the hash slot c to be checked, the index i that should be present in the hash slot, and the data item v to be made valid. A CAS at c with i as both the compare and swap values serves to atomically verify the contents of the hash slot. The resulting value should be i, but if it is not, the entire operation must be retried. If the index is correct, a write to v is performed to set the valid flag to true and the operation is complete. A special value of 0 for v indicates there is no new data to be made valid, and only a check is necessary to make the operation complete.

Before calling $finish$, it is sometimes necessary to make a new data item active (available from the index table) during a write or to change a hash slot to an index of 0 during a delete. For these situations we introduce a function called *swap* which takes three arguments in addition to the arguments to $finish$: a hash slot s to be changed, the index j it should currently contain, and the new index n to be set in the hash slot. *swap* first performs a CAS operation on s to swap j and n. If this fails, the entire operation must be retried. Otherwise, the function proceeds by calling $finish$.

3.1 Read

When a client wants to read data, it will first read the index stored at the key's primary slot in the hash table. If the slot is empty, the secondary slot must be checked. Assuming the slot is not empty, the client proceeds to read the data stored at the specified index. Because cuckoo hashing uses multiple potential indices per hash, it is possible that the data item is for a different key than the one requested and the client must therefore verify that the key stored in the data item matches the requested key. If the keys match, the corresponding value is returned to the client. Otherwise, the secondary slot must be checked.

When the primary slot does not contain the requested key, the client reads the index stored at the secondary slot in the hash table. If the slot is empty, or the key stored at the secondary slot's data item does not match the one requested, the read returns with an empty value. Otherwise, the corresponding value is returned to the caller.

3.2 Delete

Deleting a specified key k requires checking both the primary and the secondary hash slots (p and s). First, the client checks to see if the index i at s is 0, which would indicate that the delete at s is finished. When i is not 0, the client only sets s to 0 if the data item at i corresponds to k.

Once the client is finished with the secondary slot, it checks the index j stored at p. If j is already 0, or the data item at j is for a different key, the client simply calls $finish$ with arguments: s, 0, and 0. Otherwise, the client calls *swap* with arguments: p, j, 0, s, 0, and 0. This effec-

tively deletes p and verifies that the operation has completed successfully.

3.3 Write

For a write operation, the parameters are: the key k being written to, the value v it should take, and the primary and secondary slots p and s for k. After writing the key-value pair to an available data item, the operation attempts to set p to be the new data item and checks the value stored at s. If the write to p succeeds, the operation calls the *primary_write* function. Otherwise, the data stored at the primary slot must be checked to see if the active data item's key matches k. In the case of a match, the write is aborted because of a concurrent operation on the primary slot, and another attempt is made to place the data at the primary slot. A key mismatch indicates that the write must be performed at the secondary slot, which is achieved by calling *secondary_write*.

If the data is being placed at the primary slot, the client must make sure that s does not also contain a value for k. Therefore, *primary_write* must clear the secondary slot if it contains data for the same key that is being written to p. *primary_write* completes by calling either the $finish$ or *swap* function, depending on whether or not the secondary slot must be cleared, which makes the data valid for readers.

Otherwise, if the primary slot is unavailable, data must be written to the secondary slot. The client calls *secondary_write* which ensures that s is either empty or valid for the key k before completing the write. If s already contains data for another key, the write aborts and a migration operation is performed before the write is attempted again.

Algorithm 1 Write

```
 1: function WRITE(k, v, p, s)
 2:     D[n] = v                                  ▷ valid flag off
 3:     i = CAS(I[p], 0, n)
 4:     j = CAS(I[s], 0, 0)
 5:     if (i == 0) then
 6:         PRIMARY_WRITE                         ▷ primary is empty
 7:     d = D[i]
 8:     if (d.key != k) then
 9:         SECONDARY_WRITE                       ▷ primary already used
10:     r = CAS(I[p], i, n)
11:     if (r == i) then
12:         PRIMARY_WRITE                         ▷ overwrite value
13:     return Retry
14: end function
```

3.4 Migrate

If a write operation fails because both the primary and secondary slots are filled with data for different keys, then a migration must be performed to empty a slot for the write. The first migration candidate is the data in the key's primary slot. If the migration fails because the alternative slot for the data item being migrated is filled, a second migration attempt is made to relocate the data item in the secondary slot. A failure to empty the secondary slot results in a message being sent to the server, which must revoke access to its local memory and perform a chain of migrations or a table resize before allowing clients to proceed.

The input to the migration function is the key k to be migrated, the source slot l to be emptied, and the destination slot d (which is k's alternative slot). Algorithm 2 illustrates the pseudocode for performing a migration. The simplest situation is that the source slot is already empty when the migration attempt begins, in which case nothing needs to be done. In the more likely case that there is an index at the source slot, the data item is checked to ensure that the data item's key matches the key k being migrated, and a new data item is created with a copy of k's current value. The destination slot can now be set to the copied data item and the source slot cleared (assuming, of course, that the destination slot is available). As usual, if any of the slots are not available (for example, if they are being modified by another client), the operation is aborted.

Algorithm 2 Migration

```
 1: function MIGRATE(k, l, d)
 2:     i = CAS(I[l], 0, 0)
 3:     if (i == 0) then
 4:         return Done                  ▷ source is already empty
 5:     d = D[i]
 6:     if (d.key != k) then
 7:         return Abort                 ▷ source key does not match
 8:     D[n] = d                                  ▷ valid flag off
 9:     j = CAS(I[d], 0, n)
10:     if (j != 0) then
11:         return Abort                 ▷ destination not empty
12:     r = CAS(I[l], i, 0)
13:     if (r != i) then
14:         return Abort                          ▷ source changed
15:     Make D[n] valid                           ▷ migration success
16:     return Done
17: end function
```

4. ANALYSIS

Using the guarantee that RDMA operations within a connection are completed in FIFO order, it is possible to pipeline RDMA operations in certain cases. For example, the first 3 RDMA operations of the write function have no dependencies on one another, and so they can all be issued without waiting for the previous operation to finish. The advantage of pipelining is that the roundtrip times are overlapped, thereby reducing the total latency.

To provide a simple assessment of Nessie we calculate the expected number of roundtrips needed, for a range of load factors on the hash table, when a single writer is attempting to add a new key into the key-value store. To simplify the calculation, we assume that there are no other writers, the RDMA operations are pipelined whenever possible, and it is always possible for the client to complete the write (without contacting the server). The results are shown in Table 1.

Load Factor	Roundtrips
0.1	2.5
0.25	3.5
0.5	5.8
0.75	9.9
0.9	13.3

Table 1: Expected Number of Roundtrips

Notice that when the hash table is 75% loaded the expected latency to complete the entire write operation is approximately 100 microseconds, which is less than the latency to complete a single TCP roundtrip. Of potential concern is that we have assumed the write is able to succeed without contacting the server. In other words, the client is able to migrate a key occupying either its primary or secondary slot into an alternative empty slot. At a load of 0.75, there is a 70% chance that this will be possible, and further additions to the migration function would allow the client to handle more complex situations, such as migrations triggered by migrations, without having to contact the server.

For many deployment scenarios, the size of the index table would be insignificant in comparison to the size of the actual data. Therefore, a lower load factor such as 0.5 or 0.25 is both reasonable and expected. Under these circumstances, Nessie is able to perform writes with significantly fewer roundtrips.

5. DISCUSSION

Nessie makes use of a myriad of different techniques, such as RDMA CAS operations, key-specific version numbers, pre-assigned free memory regions, and incomplete table entries, to essentially provide serializability without involving the storage host's CPU. Although these techniques are relatively efficient, they significantly increase the complexity of the design. Simple design modifications, such as increasing the number of cuckoo hash locations per key or inserting a default value when reading a non-existent key, require non-trivial changes to how the system uses these techniques. Reasoning about the correctness of the design is also difficult. We only provide simple constructive reasoning for the correctness of our design; a formal proof is beyond the scope of this paper because of the complexity of the design.

Given that these techniques are used to provide serializability, an alternative lock-free hash table design is to use hardware transactional memory (HTM). HTM is becoming increasingly available; the latest Intel Haswell processors already provide support for HTM with three new explicit instructions: XBEGIN, XEND, and XABORT. There has been recent work [8, 16] on using these instructions to implement efficient in-memory storage systems. Unfortunately, although RDMA memory accesses can safely interact with hardware transactions since RDMA operations respect cache coherence, current NICs cannot actively initiate a transaction. Therefore, with current hardware, it is not possible to use HTM to simplify a hash table design that only uses one-sided RDMA operations to perform read and write requests. Adding support for remote hardware transaction memory (RHTM) to RDMA requires changes that, even with significant industry support, may take several years to come to market.

However, in a rack-scale system, proprietary changes to the hardware and can be quickly added if there is application demand. Adding RHTM support would at least require introducing high-speed cache to NICs to store uncommitted writes, and remote versions of XBEGIN, XEND and XABORT. Although these are non-trivial hardware changes, they are within the scale of architectural changes that are common in specialized high-performance rack-scale systems. Therefore, in the next section, we explore using RHTM to implement a hash table and outline changes to Nessie to take advantage of RHTM.

5.1 Hash Table Design Using RHTM

A naive approach to using RHTM in a hash table design is to surround both read and write operations with remote XBEGIN and XEND operations. This approach follows the philosophy of transactional memory, but is unlikely to work well in any hardware transactional memory implementation. This is because HTM implementations generally use high-speed cache to buffer uncommitted writes; performing many large, concurrent RDMA reads and writes will lead to cache eviction on the NIC, which in turn increases the transaction abort rate. Therefore, even with RHTM, it is still important to use self-verifying data-structures and pre-assign free memory regions to clients. This will allow the large and highly concurrent reads and writes to be non-transactional, and only the reads and writes to the index table need to be inside a transaction.

It is relatively straightforward to modify Nessie to use RHTM. The basic structure of the hash table remains the same with a separate index and data table. Reads and writes to the data table also remain the same. However, there is no longer a need for a valid flag in the self-verifying data structure. This is because multiple modifications to the index table can now be performed atomically using a transaction, which obviates the need for a mechanism to delay revealing an uncommitted write. Entries in the index table no longer require version numbers since the ABA problem cannot occur when using transactions; any modifications to an entry will be detected and cause the transaction to abort. More importantly, the terminating CAS instructions in Nessie operations, which detect conflicting modifications by another request, are also unnecessary.

These changes not only reduce the complexity of the design, but they also improve performance by immediately aborting a transaction when there is a conflict, and reducing the number of RDMA round trips by eliminating CAS operations used only for conflict checking.

6. CONCLUSIONS

The prevalence of RDMA in high-performance NICs provides system designers with a tremendous opportunity to re-evaluate current performance-critical systems, such as distributed in-memory hash table, and rebuild them with RDMA to reduce both latency and system overhead. In this paper, we build on past work to design Nessie, a low-latency cuckoo hash table that allows remote clients to not only perform read requests using one-sided RDMA operations, but also write requests. Nessie makes use of techniques such as self-verifying data-structures, atomic RDMA CAS operations, and incomplete table entries to ensure that requests are serialized without building or using locks. Our analytic results show that with a reasonable hash table load factor, Nessie can complete most read requests in only two roundtrips and most write requests in under 5.8 roundtrips. Finally, in order to reduce the complexity of our hash table design, we propose extending RDMA with RHTM and outline a simpler version of Nessie that uses RHTM.

7. ACKNOWLEDGMENTS

We thank the Natural Sciences and Engineering Research Council of Canada and GO-Bell scholarship for funding. Additionally, this work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo.

8. REFERENCES

[1] ABA problem. `http://en.wikipedia.org/wiki/ABA_problem`, January 2014.

[2] Rdma over converged ethernet (roce). `http://www.mellanox.com/page/products_dyn?product_family=79`, March 2014.

[3] A. Dragojević, D. Narayanan, and M. Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, Seattle, WA, April 2014.

[4] B. Fan, D. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, Lombard, IL, April 2013.

[5] J. Huang, X. Ouyang, J. Jose, M. Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. Panda. High-Performance Design of HBase with RDMA over InfiniBand. In *Proceedings of the 26th International Parallel and Distributed Processing Symposium*, Shanghai, China, May 2012.

[6] N. Islam, M. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. Panda. High Performance RDMA-based Design of HDFS over InfiniBand. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT, November 2012.

[7] B. Li, P. Zhang, Z. Huo, and D. Meng. Early Experiences with Write-Write Design of NFS over RDMA. In *Proceedings of the IEEE International Conference on Networking, Architecture, and Storage*, Zhang Jia Jie, Hunan, China, July 2009.

[8] X. Li, D. Andersen, M. Kaminsky, and M. Freedman. Highly Concurrent Hash Tables Need More Than Naive Use of Hardware Transactional Memory. In *Proceedings of the European Conference on Computer Systems*, Amsterdam, The Netherlands, April 2014.

[9] J. Liu, W. Jiang, P. Wyckoff, D. Panda, D. Aston, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, Santa Fe, NM, April 2004.

[10] J. Liu, J. Wu, and D. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3):167–198, June 2004.

[11] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the USENIX Annual Technical Conference*, San Jose, CA, June 2013.

[12] R. Pagh and F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(3):122–144, May 2004.

[13] S. M. Rumble, D. Ongaro, and R. Stutsman. It's Time for Low Latency. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems*, Napa, CA, May 2011.

[14] G. Shipman, T. Woodall, R. Graham, A. Maccabe, and P. Bridges. Infiniband Scalability in Open MPI. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, April 2006.

[15] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2012.

[16] Z. Wang, H. Qian, J. Li, and H. Chen. Using Restricted Transactional Memory to Build a Scalable In-Memory Database. In *Proceedings of the European Conference on Computer Systems*, Amsterdam, The Netherlands, April 2014.